1/7

MAGED M. MICHAEL et al

YOR920030268US1 (DSJ) 8728-632

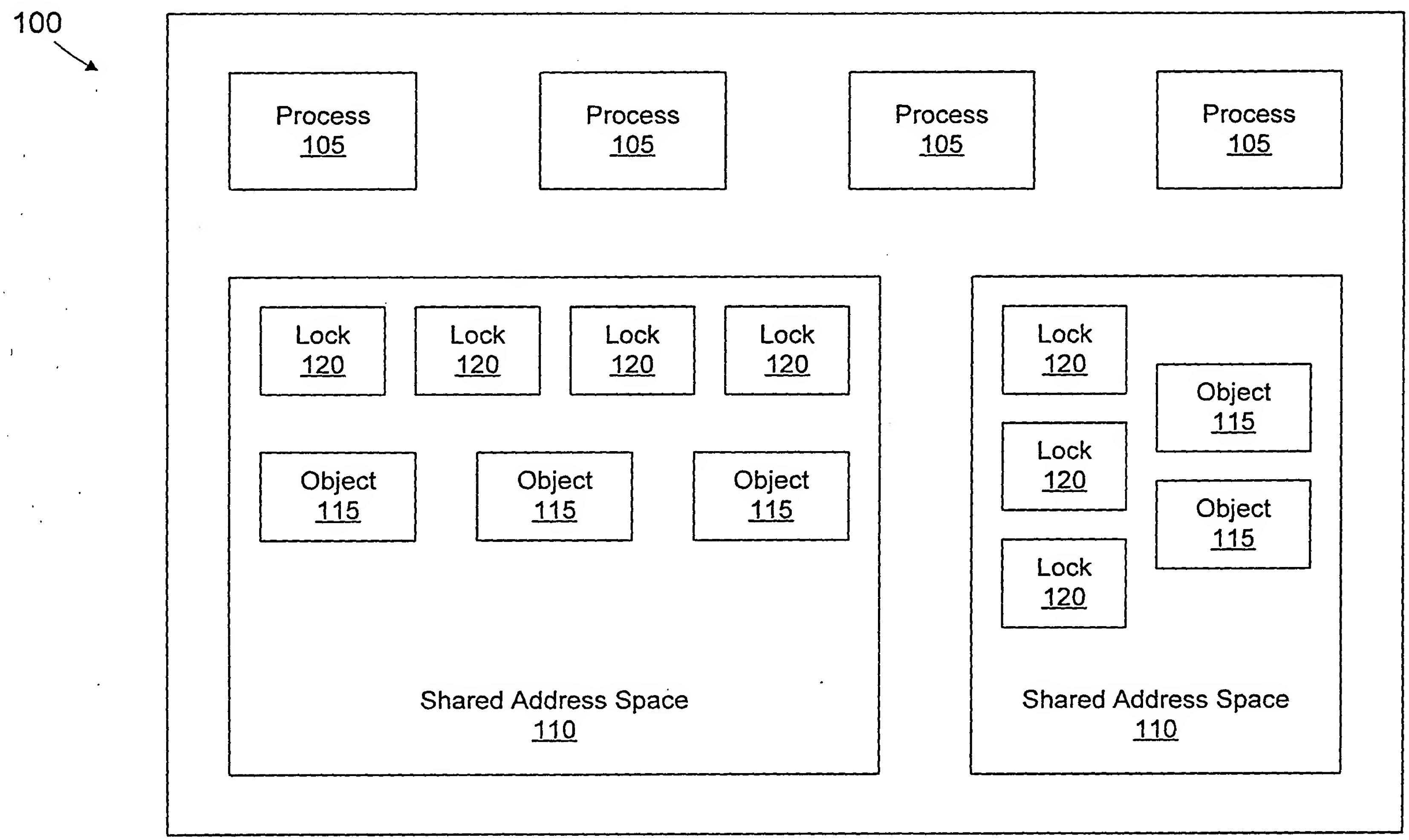

Figure 1
(Prior Art)

2/7
YOR920020268US2 (8728-632)

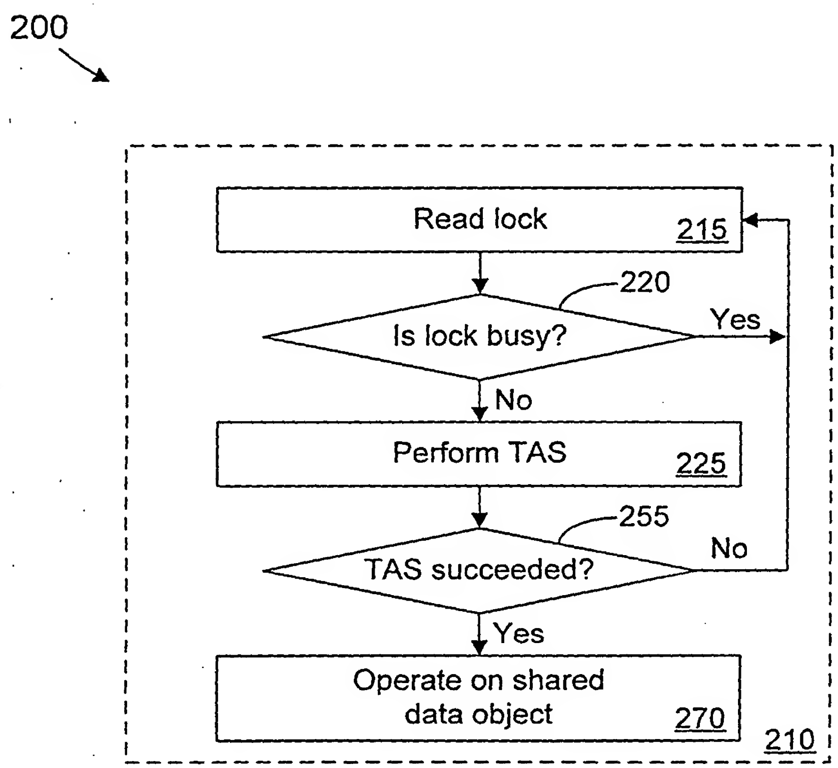

Figure 2A
(Prior Art)

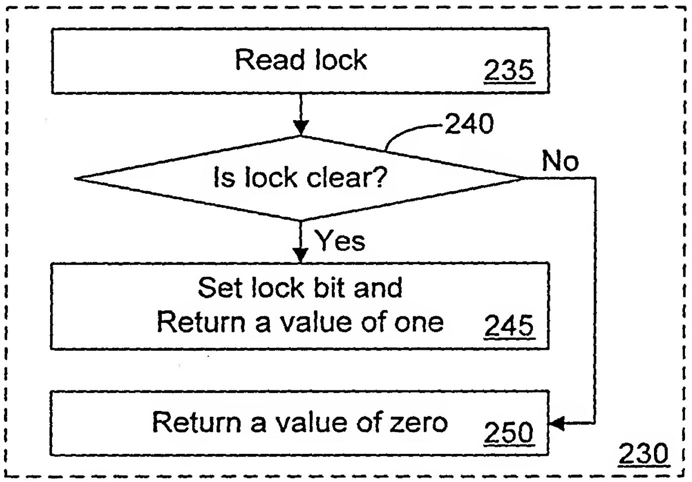

Figure 2B
(Prior Art)

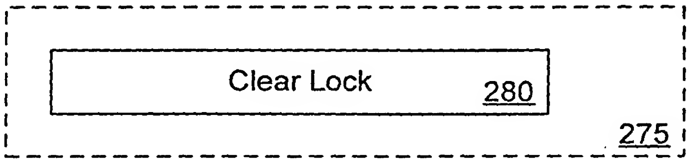

Figure 2C
(Prior Art)

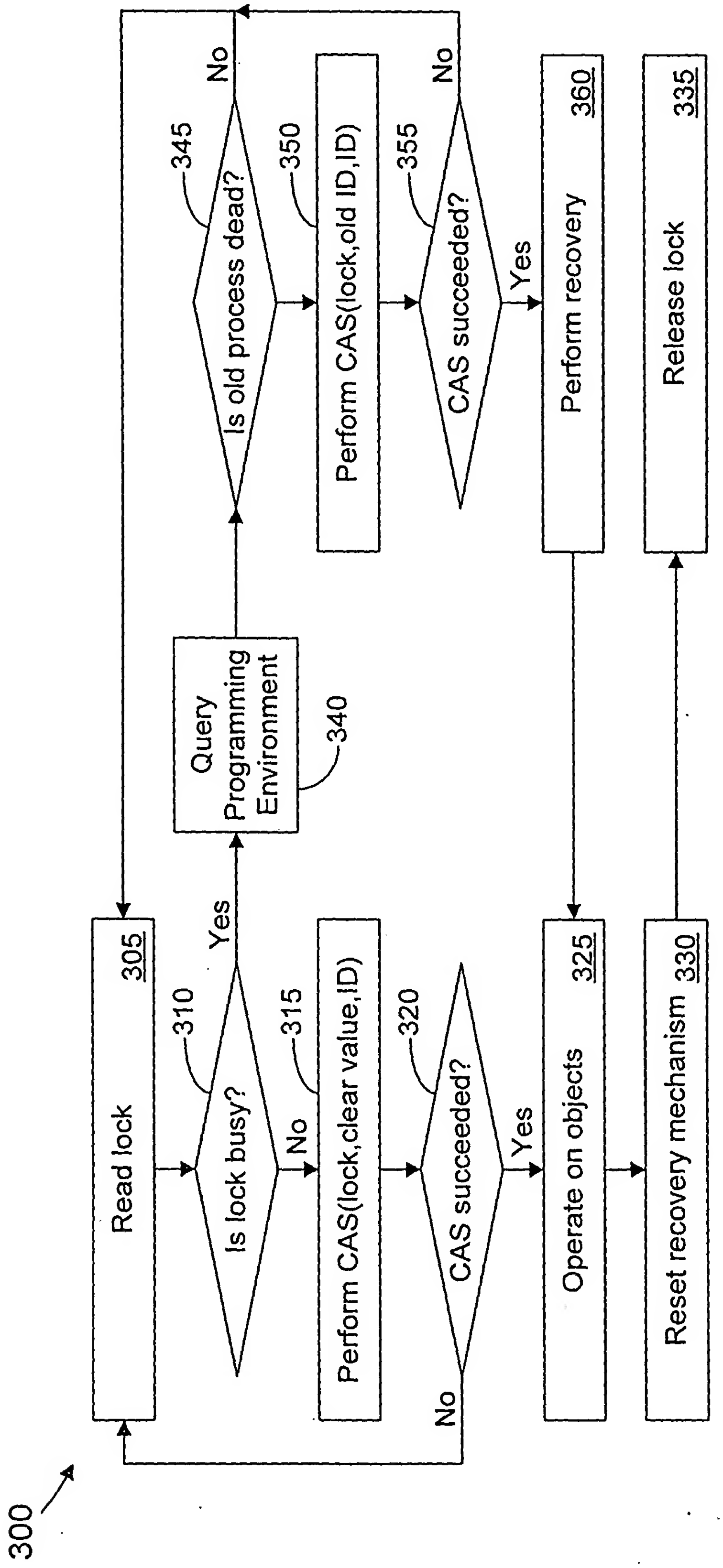
300
Read lock
305
Is lock busy?
310
No
Yes
Perform CAS(lock,clear value,ID)
315
CAS succeeded?
320
No
Yes
Operate on objects
325
Reset recovery mechanism
330
Query Programming Environment
340
Is old process dead?
345
No
Perform CAS(lock,old ID,ID)
350
CAS succeeded?
355
No
Yes
Perform recovery
360
Release lock
335

Figure 3

```
// Lock variables
LockHolder : ProcessIdType;
LastChecked : TimeType;

AcquireLock (p : ProcessIdType)
        while true {
A1:       if (holder ← LockHolder) = NULL {
A2:           if CAS(&LockHolder, NULL, p)
A3:              {LastChecked ← Time(); return; }
          } else {
A4:         last ← LastChecked ;
A5:         time ← Time();
A6:         if time - last < THRESHOLD continue;
A7:         if ¬CAS(&LastChecked, last , time) continue;
A8:         if QueryOS(holder) = ALIVE continue;
A9:         if CAS(&LockHolder, holder, p)
A10:           {UserRecovery (. . .); return; }
          }
        }

ReleaseLock ()
R1:  ResetRecovery (. . .);
R2:  LockHolder ← NULL;
```

Figure 4

5/7

YOR920030268US1 (8728-632)

```
structure NodeType
    Pid : ProcessIdType;
    Status : ({WAITING, HASLOCK, FAILED}, TagType);
    Next : (*NodeType, TagType);
    LastChecked : TimeType;
Head, Tail : (*NodeType, TagType) initially NULL;
LockHolder : *NodeType initially NULL;

AcquireLock (node : *NodeType)
A1: Enqueue(node);
A2: old_head ← Head.Data; ptr ← old_head;
    while true {
A3:   if WaitForSignal () = HASLOCK {
A4:     node^.LastChecked ← Time();
A5:     LockHolder ← node; return;
      }
      // Timeout is detected.
A6:   holder ← LockHolder,
A7:   if (head ← Head.Data) ← old_head {
A8:     old_head ← head, ptr ← head,
A9:   } elseif ptr ≠ node {
A10:    ptr ← ptr^.Next .Data,
      } else {
A11:    old_head ← head, ptr ← head,
A12:    if holder ≠ NULL ∧ holder ≠ head {
A13:      if ¬ProcessFailed(holder, head) continue;
A14:      if holder ≠ LockHolder continue;
        }
A15:    if UsurpLock (node, head) return;
      }
    }

ReleaseLock (node : *NodeType)
R1: ResetRecovery (. . .);
R2: next ← Dequeue(node);
R3: if next ≠ NULL   <status, t> ← next^.Status;
R4: CAS(&LockHolder, node, NULL);
R5: if next ≠ NULL
      CAS(&next^.Status, <status, t>, <HASLOCK, t + 1>);

ProcessFailed(ptr , head : *NodeType) : boolean
    while true {
Q1:   if ptr^.Status = <FAILED, 0> return true;
Q2:   if head ≠ Head.Data return false;
Q3:   last ← ptr^.LastChecked ;
Q4:   time ← Time();
      if time - last ≤ THRESHOLD continue;
Q5:   if □CAS(ptr^.LastChecked , last , time) continue;
Q6:   if QueryOS(ptr^.Pid) = ALIVE return false;
Q7:   node^.Status ← <FAILED, 0>; return true;
    }

WaitForSignal (node : *NodeType) : {HASLOCK,
TIMEOUT}
W1: last ← Time();
    while true {
W2:   status ← node^.Status.Data,
W3:   if status = HASLOCK return HASLOCK;
W4:   time ← Time();
W5:   if time - last > THRESHOLD return TIMEOUT;
    }

UsurpLock (node, head : *NodeType) : boolean
U1: ptr□ head,
    while ptr ≠ node {
U2:   if ¬ProcessFailed(ptr, head) return false,
U3:   ptr ← ptr^.Next .Data,
U4:   if head ≠ Head.Data return false;
    }
U5: Head ← <node, Head.Tag + 1>;
U6: if node ≠ head UserRecovery (. . .);
U7: node^.Status ← <HASLOCK, node^.Status.Tag + 1>;
U8: LockHolder ← node; return true;

Enqueue(node : *NodeType)
E1: node^.Next ← <NULL, node^.Next .Tag + 1>;
    while true {
E2:   <tail, $t_T$> ← Tail ;
E3:   <head, $t_H$> ← Head,
      if tail ≠ NULL ∧ head ≠ NULL {
E4:     <next, $t_N$> ← tail^.Next ;
        if next = NULL {
E5:       if Tail ≠ <tail, $t_T$> continue;
E6:       node^.Status ← <WAITING, node^.Status.Tag + 1>:
E7:       if ¬CAS(&tail^.Next , <NULL, $t_N$>, <node, $t_N$ + 1>)
            continue;
E8:       CAS(&Tail, <tail, $t_T$ >, <node, $t_T$ + 1>);
          return;
        } elseif next = DEQUEUED {
E9:       if ¬CAS(&Tail , <tail, $t_T$ >, <NULL, $t_T$ + 1>)
            continue;
E10:      CAS(&Head, <head, $t_H$>, <NULL, $t_H$ + 1>);
        } else
E11:      CAS(&Tail , <tail, $t_T$ >, <next, $t_T$ + 1>);
      } elseif tail = NULL ∧ head = NULL {
E12:    node^.Status ← <HASLOCK, node^.Status.Tag + 1>;
E13:    if CAS(&Tail , <NULL, $t_T$>, <node, $t_T$ + 1>) return;
E14:  } elseif Tail = <tail, $t_T$>
E15:    CAS(&Head, <head, $t_H$>, <tail, $t_H$ +1>);
    }

Dequeue(node : *NodeType) :
*NodeType
D1: <next, $t_N$> ← node^.Next ;
D2: <tail, $t_T$> ← Tail ;
      if node = tail {
        if next = NULL {
D3:       if CAS(&node^.Next ,
                <NULL, $t_N$>,
             <DEQUEUED, $t_N$ + 1>) {
D4:         CAS(&Tail , <node, $t_T$>,
<NULL, $t_T$ + 1>);
D5:         $t_H$ ← Head.Tag ;
D6:         CAS(&Head, <node, $t_H$>,
<NULL, $t_H$ + 1>);
            return NULL;
          }
D7:       <next, $t_N$>□ node^.Next ;
        }
D8:     CAS(&Tail , <node, $t_T$>, <next,
$t_T$ + 1>);
      }
D9: Head ← <next, Head.Tag + 1>;
return next;
```

Figure 5

```
structure NodeType
  Pid : ProcessIdType;
  Status : ({WAITING, HASLOCK, FAILED, REMOVED,
                     TO_BE_REMOVED}, TagType);
  Next : (*NodeType, TagType);
  LastChecked : TimeType;
  Signal : *NodeType;
  Ack : boolean;
Head, Tail : (*NodeType, TagType) initially NULL;
LockHolder : *NodeType initially NULL;

PS-AcquireLock(node : *NodeType)
a1: repeat until TryLock (node) = HASLOCK;

PS-ReleaseLock(node : *NodeType)
r1: ResetRecovery (. . .);
r2: next ← Dequeue(node);
    while next = NULL {
r3:   <status, t> ← next^.Status;
r4:   If CheckPreemption(node, next) = ACTIVE {
r5:     CAS(&LockHolder, node, NULL);
r6:     CAS(&next^.Status, <status, t>, <HASLOCK, t + 1>);
r7:     return;
      }
r8:   next^.Status ← <TO_BE_REMOVED, t+ 1>;
r9:   ptr ← Dequeue(next);
r10:  CAS(&next^.Status, <TO_BE_REMOVED, t+ 1>, <REMOVED, t + 2>);
      next ← ptr ;
    }
r11: CAS(&LockHolder, node, NULL);

CheckPreemption(node, next : *NodeType) : {ACTIVE, PREEMPTED}
c1: node^.Ack ← false;
c2: next^.Signal ← node;
c3: for i ← 0 to PREEMPTION_THRESHOLD
c4:   If node^.Ack return ACTIVE;
c5: return PREEMPTED;

TryLock (node : *NodeType) : {HASLOCK, REMOVED}
t1: Enqueue(node);
t2: old_head ← Head.Data; ptr ← old head;
    while true {
t3:   If (status ← PS-WaitForSignal()) = HASLOCK {
t4:     node^.LastChecked ← Time();
t5:     LockHolder ← node; return HASLOCK;
      }
t6:   elseif status = REMOVED return REMOVED;
t7:   holder ← LockHolder ;
t8:   head ← Head.Data;
t9:   <status, t> ← node^.Status;
      If status = TO_BE_REMOVED ∧ node ≠ head
t10:    {node^.Status ← <REMOVED, t+ 1>; return REMOVED; }
t11: If head ≠ old head {
t12:   old_head ← head; ptr ← head;
t13: } elseif ptr ≠ node {
t14:   ptr ← ptr^.Next .Data;
     } else {
t15:   old_head ← head; ptr ← head;
t16:   If holder ≠ NULL ∧ holder ≠ head {
t17:     If ¬ProcessFailed(holder, head) continue;
t18:     If holder ≠ LockHolder continue;
       }
t19:   If UsurpLock (node, head) return HASLOCK;
     }
   }

PS-WaitForSignal(node : *NodeType) : {HASLOCK, REMOVED, TIMEOUT}
w1: last ← Time();
w2: node^.Signal ← NULL;
    while true {
w3:   If (ptr ← node^.Signal) = NULL {
w4:     ptr^.Ack ← true;
w5:     node^.Signal ← NULL;
      }
w6: status ← node^.Status.Data;
w7: If status ∈ {HASLOCK, REMOVED} return status;
w8: time ← Time();
w9: If time − last > THRESHOLD return TIMEOUT;
  }
```

Figure 6

YDR920030268US1 (8728-632)

```
// MAX is the maximum number of updates in a CS
structure LogType
        count : 0..MAX; // initially 0
        addr[MAX] : pointer;
        prev[MAX] : ValueType;

WriteAndLog(addr : pointer, v : ValueType)
        log.prev[log.count ] ← *addr;
        log.addr[log.count ] ← addr;
        log.count ← log.count + 1;
        *addr ← v;

UserRecovery ()
        for i ← log.count - 1 downto 0
                *log.addr[i] ← log.prev [i];
        log.count ← 0;

ResetRecovery ()
        log.count ← 0;
```

Figure 7